

TAPNext++: What’s *Next* for Tracking Any Point (TAP)?

Supplementary Material

A. Frequently Asked Questions

Q: Is the improved performance on PointOdyssey simply due to training on the PointOdyssey dataset?

To verify generalization, we trained a variant of TAPNext++ exclusively on Kubric-1024 and DynHumans, excluding PointOdyssey from training entirely. As shown in Tab. 5, this zero-shot variant still outperforms BootsTAPNext-B by 38.3 percentage points in survival rate and exceeds Track-On by 3.8 points, demonstrating that the long-sequence training recipe generalizes to out-of-domain videos beyond the PointOdyssey distribution.

Table 5. **Training without PointOdyssey.** Evaluation results on PointOdyssey (PO) with and without including PO in the training mix.

Method	$\delta^{avg} \uparrow$	Survival \uparrow	MTE \downarrow
CoTracker3	44.5	56.3	20.7
Track-On	35.4	47.5	33.5
BootsTAPNext-B (Baseline)	9.9	13.0	92.1
Ours (w/ PO)	52.6	67.9	13.4
Ours (w/o PO)	37.3	51.3	26.4

Q: Does the roll augmentation give TAPNext++ an unfair advantage on AJ_{RD}?

We isolate the effect of roll augmentation in Tab. 3. While it benefits RoboTAP and Kinetics, it slightly reduces scores on PointOdyssey and DAVIS. Crucially, even *without* roll augmentation, our model remains state-of-the-art on DAVIS (AJ = 65.3) and PointOdyssey ($\delta^{avg} = 52.0$, Survival = 68.6, MTE = 16.4, AJ_{RD} = 22.3). The roll augmentation is a principled, task-specific training technique that any method can equally adopt; it is not a post-hoc advantage restricted to TAPNext++.

Q: Does TAPNext++ claim to solve infinite-length point tracking?

No. We explicitly target long but finite sequences and do not claim that performance degradation is fully eliminated at arbitrary sequence lengths. Our contribution is to demonstrate that such degradation is *not* an inherent architectural limitation of *linear* recurrent units—in contrast to standard non-linear recurrent architectures that suffer from vanishing gradients. By extending the effective tracking range from ~ 150 frames to over 4000 frames without any architectural change, we provide new insights into the practical scalability of linear recurrent units for point tracking.

Q: Is TAPNext++ merely a “bag of tricks” with limited technical novelty?

We argue that system-algorithmic co-design is a fundamental research contribution, particularly in the context of scalable training. Concretely: (1) the distributed parallel scan for SSM blocks is not a standard engineering detail but the technical enabler that makes end-to-end training on 1024-frame sequences feasible on multiple GPUs—without it, naïve sequence parallelism would require $\mathcal{O}(N)$ sequential communication rounds instead of $\mathcal{O}(\log N)$; (2) the roll augmentation is a principled, task-specific design directly addressing the well-known failure mode of points exiting and re-entering the frame; (3) AJ_{RD} fills a concrete gap in the evaluation literature by directly measuring post-reappearance tracking quality, which no prior metric captures.

Q: How does Kubric-1024 differ from prior Kubric datasets?

Kubric-1024 required a fundamental redesign to sustain scene dynamics over 1024 frames. In prior Kubric variants, object motion dissipates quickly due to passive physics. We address this by introducing “velocity bumps”—random linear and angular impulses applied whenever an object’s speed drops below a threshold, with a slight bias toward the scene origin to keep objects within the camera’s field of view throughout the full sequence. We additionally apply sinusoidal noise to both the camera position and look-at point to simulate natural camera jitter, preventing the model from overfitting to the unnaturally smooth trajectories present in prior synthetic datasets.

Q: Why does the high-resolution (512×512) variant sometimes underperform the 256×256 model in Tab. 1?

To ensure a fair comparison, the high-resolution variant was fine-tuned using the same training recipe as the 256×256 model rather than a separately tuned configuration. The optimal hyperparameters for higher-resolution fine-tuning—including learning rate, schedule length, and batch composition—likely differ and were not optimized in this work. The results in Tab. 1 should therefore be treated as a conservative lower bound on the potential of the 512×512 model.

B. Long-Sequence Training with Sequence Parallelism

Training TAPNext on long sequences ($T = 1024$ frames or more) requires fitting large computation graphs and intermediate activations into memory, which exceeds single-device limits. To enable end-to-end training on such sequences, we adopt Sequence Parallelism (SP). The input sequence is partitioned into N chunks along the tempo-

ral dimension, with each chunk processed by a dedicated GPU. This strategy requires communicating $\mathcal{O}(T^2)$ bits for attention-based models via context parallelism. At the same time the linear recurrent nature of the State Space Models (SSMs) allows performing forward and backward propagation with only $\mathcal{O}(\log T)$ sequential and $\mathcal{O}(T \log T)$ total communication bits as we show below.

The Real-Gated Linear Recurrent Unit (RG-LRU) used in TAPNext, like other SSMs, is defined by a linear recurrence relation of the form:

$$h_t = a_t \odot h_{t-1} + x_t, \quad (1)$$

where $h_t, a_t, x_t \in \mathbb{R}^D$ are the hidden state, recurrence parameters, and input at time t , respectively, and \odot denotes element-wise multiplication. This recurrence can be expressed as an associative binary operator. First, if $h_1 = a_1 \odot h_0 + x_1$ and $h_2 = a_2 \odot h_1 + x_2$, then $h_2 = (a_2 \odot a_1) \odot h_0 + (a_2 \odot x_1 + x_2)$. This allows us to define an associative binary operator \oplus :

$$(a_2, x_2) \oplus (a_1, x_1) = (a_2 \odot a_1, a_2 \odot x_1 + x_2). \quad (2)$$

Because this operator is associative, we can compute the recurrence using a parallel scan algorithm, rather than a sequential loop. We leverage this property to implement a *distributed parallel scan* for RG-LRU during the forward and backward passes, enabling efficient training on distributed sequence chunks. Our distributed scan operates in three phases:

1. **Local Scan:** Each GPU $j \in \{0, \dots, N-1\}$, holding sequence chunk $\{(a_t^j, x_t^j)\}_{t=0}^k$, computes a local parallel scan assuming its initial hidden state h_{in}^j is zero. This produces a preliminary local output y'^j and a chunk summary $S_j = (\alpha_j, \chi_j)$, where $\alpha_j = a_k^j \odot \dots \odot a_0^j$ and $\chi_j = y_k'^j$. This summary S_j represents the affine transformation of the chunk: $h_{\text{out}}^j = \alpha_j \odot h_{\text{in}}^j + \chi_j$. This phase is executed in parallel across all N GPUs.
2. **Cross-GPU Prefix Scan:** To find the correct input state h_{in}^j for each chunk $j > 0$, we must compute $h_{\text{in}}^j = h_{\text{out}}^{j-1}$. This requires composing the transformations of all preceding chunks: $h_{\text{in}}^j = (\alpha_{P_{j-1}} \odot h_{\text{start}}) + \chi_{P_{j-1}}$, where $P_{j-1} = S_{j-1} \oplus S_{j-2} \oplus \dots \oplus S_0 = (\alpha_{P_{j-1}}, \chi_{P_{j-1}})$ and h_{start} is the initial state of the entire sequence (typically zero). Computing all prefix compositions $\{P_0, P_1, \dots, P_{N-1}\}$ is a standard prefix scan problem over the set of summaries $\{S_0, S_1, \dots, S_{N-1}\}$ using the operator \oplus . This is solved efficiently across N GPUs using tree-based algorithms (e.g., recursive doubling) with $\mathcal{O}(\log N)$ communication rounds, rather than $\mathcal{O}(N)$ rounds required by sequential propagation.
3. **Local Correction:** Once each GPU j receives its correct initial state h_{in}^j from Phase 2, it corrects its preliminary output y'^j by adding the contribution from h_{in}^j : $y_t^j =$

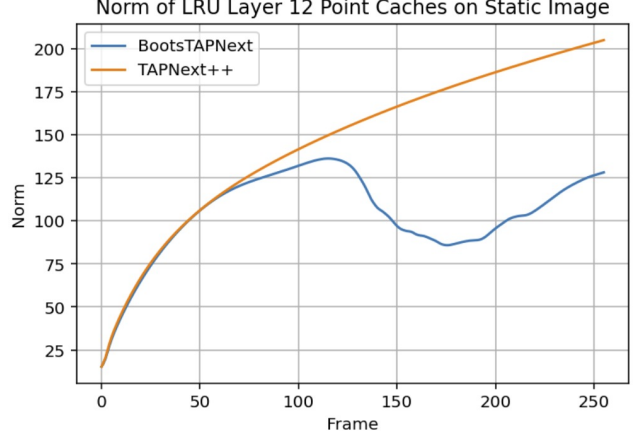


Figure 8. **LRU State Norm Growth on re-feeding a Single Image.** BootsTAPNext shows unstable LRU state after 100 frames. TAPNext++ has a monotonically growing state.

$y_t'^j + c_t^j \odot h_{\text{in}}^j$, where $c_t^j = a_t^j \odot a_{t-1}^j \odot \dots \odot a_0^j$. This phase is also performed in parallel across all GPUs.

The backward pass follows a similar distributed scan pattern to propagate gradients efficiently across GPUs.

In addition to that, we also implement sequence parallelism for temporal causal convolution layers. Unlike the linear recurrence, each step of the sequence may only depend on a constant number of past steps in forward pass for temporal convolution (or future steps during backward pass). Therefore, we only need to do steps 1 and 2 and for the latter we can do one linear chain of communications in parallel.

This method allows TAPNext to be trained end-to-end on sequences significantly longer than those trainable on a single device, with only a logarithmic increase in communication overhead with respect to the number of GPUs.

C. Dynamic Model Depth

Prior work[30] has shown that TAPNext-like architectures can be trained using a layer-wise supervision strategy, meaning that each layer is trained using its own loss. As a result, the model supports dynamic inference depth: it can be executed with fewer layers at test time without requiring retraining. Although the original architecture uses twelve layers, we observed that the predictions produced after only eight layers are already comparable in quality to those obtained after all twelve. This allows one to reduce memory consumption and further increase the achievable inference speed (FPS). Note that in this work, all experiments were executed using the full twelve layers.

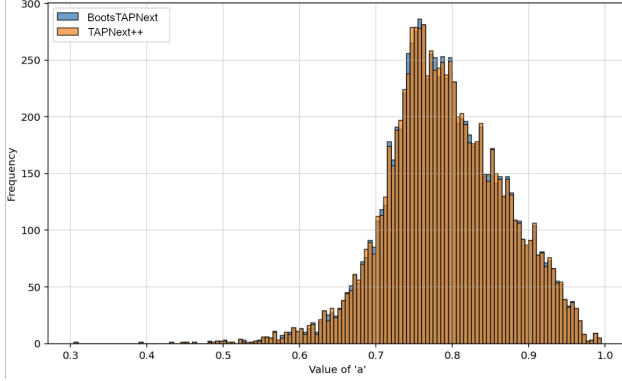


Figure 9. **Eigenvalue Distribution.** Distribution of eigenvalues a . Almost no change can be observed between BootsTAPNext and TAPNext++.

D. LRU State analysis

We analyze the temporal behavior of the LRU state associated with the point-token stream in the twelfth layer. To isolate the intrinsic LRU dynamics from changes caused by varying inputs, we repeatedly feed the same static image into the network and track the norm of the resulting cache state over time. As shown in Fig. 8, the original BootsTAPNext exhibits irregular and unstable cache growth after approximately 100 frames, whereas TAPNext++ produces a smooth and nearly monotonic evolution of the cache norm. Also note that both network show equal state growth in the beginning of the video. Importantly, even with a constant input image, changes in the LRU state are expected: the LRU update rule operates directly on the current activation and does not explicitly enforce invariance over repeated inputs.

To better understand this behavior, we further inspect the distribution of the learned eigenvalues a that parametrize the LRU update (cf. Fig. 9). Surprisingly, we do not observe a pronounced shift toward eigenvalues near one in TAPNext++, which would correspond to stronger long-term memory retention. This suggests that the improved stability of TAPNext++ does not arise from extending the effective memory horizon of the LRU block. Instead, the model may have learned to regulate or compensate for long-sequence LRU dynamics elsewhere in the architecture—potentially through more stable interactions between point- and patch-token pathways or through modifications in downstream attention layers.